Embedded.com

Born to Fail

By Jack Ganssle, Embedded Systems Programming
Dec 12 2002 (10:11 AM)
URL: http://www.embedded.com/showArticle.jhtml?articleID=9900877

Systems fail, and sometimes no one is around to reset them before something worse happens. That's why watchdog timers matter.

Launched in January 1994, NASA's Clementine spacecraft spent two successful months mapping the moon before leaving lunar orbit to head towards near-Earth asteroid, Geographos.

A dual-processor Honeywell 1750 subsystem handled telemetry and various spacecraft functions. Though the 1750 could control Clementine's thrusters, it did so only in emergency situations; routine thruster operations were under ground control.

On May 7, the 1750 experienced a floating-point exception. This wasn't unusual; some 3,000 prior exceptions had been detected and handled properly. But immediately after the May 7 event, downlinked data started varying wildly and nonsensically. Then the data froze. Controllers spent 20 minutes trying to bring the system back to life by sending software resets to the 1750; all were ignored. A hardware reset command finally brought Clementine back on-line.

Alive, yes, even communicating with the ground, but with virtually no fuel left.

The evidence suggests that the 1750 locked up, probably due to a software crash. While hung, the processor turned on one or more thrusters, dumping fuel and setting the spacecraft spinning at 80 RPM. In other words, it appears the code ran wild, firing thrusters it should never have enabled. They kept firing until the tanks ran nearly dry and the hardware reset closed the valves. The mission to Geographos was abandoned.

Designers had worried about this sort of problem and implemented a software thruster timeout. That failed, of course, when the firmware hung.

The 1750's built-in watchdog timer (WDT) hardware was not used, over the objections of the lead software designer. With no automatic reset, the mission's success rested in the abilities of the controllers on Earth to detect problems quickly. For the lack of a few lines of watchdog code, the mission was lost.

Though such a fuel dump had never occurred on Clementine before the May 7 meltdown, hardware resets from the ground had been required roughly 16 times. One might also wonder why some 3,000 previous floating-point exceptions were part of the mission's normal firmware profile.

Not surprisingly, the software team wished they had indeed used the watchdog and not implemented the thruster timeout in firmware. They also noted, though, that a normal, simple watchdog might not have been robust enough to catch the failure mode.

Contrast this with Pathfinder, a mission whose software also famously hung, but which was saved by a reliable watchdog. The software team found and fixed the bug, uploading new code to a target system 40 million miles away, enabling an amazing scientific mission on Mars.

Watchdogs are our fail-safe, our last line of defense, an option taken only when all else fails, right? Actually, these missions and many others suggest to me that WDTs are not emergency outs, but integral parts of our systems. The WDT is as important as main() or the runtime library; it's an asset that is likely to be used, and maybe used a lot.

Space is a hostile environment, of course, with high intensity radiation fields, thermal extremes, and vibrations we'd never see on Earth. Do we have these worries when designing Earth-bound systems?

Maybe we do. Intel revealed that the McKinley processor's ultra-fine design rules and huge transistor budget means cosmic rays can flip on-chip bits. The Itanium 2 processor, also sporting an astronomical transistor budget and small geometry, includes an on-board system management unit to handle transient hardware failures. The hardware ain't what it used to be, even if our software could be perfect.

Reality check

But too much (all?) firmware is not perfect. Consider this unfortunately true story from long-time reader Ed VanderPloeg:

The world has reached a new embedded software milestone: I had to reboot my hood fan. That's right, the range exhaust fan in the kitchen. It's a simple model from a popular North American company. It has six buttons on the front: three for low, medium, and high fan speeds, and three more for low, medium, and high light levels. Press a button once and the hood fan does what the button says. Press the same button again and the fan or lights turn off. That's it; nothing fancy. But it needed rebooting via the breaker panel.

Apparently the thing has a microprocessor to control the light levels and fan speeds, and it also has a temperature sensor to automatically switch the fan to high speed if the temperature exceeds some fixed threshold. Well, one day we were cooking dinner as usual, steaming a pot of potatoes, and suddenly the fan kicked into high speed and the lights started flashing. 'Hmm, flaky sensor or buggy sensor software,' I thought to myself.

The food happened to be done, so I turned off the stove and tried to turn off the fan, but, I suppose, it wanted things to cool off first. Fine. After ten minutes or so the fan and lights turned off on their own. I then went to turn on the lights, but instead they flashed continuously, with the flash rate depending on the brightness level I selected.

Just for fun I tried turning on the fan too, but all three speed buttons produced only high speed. 'What 'smart' feature is this?' I wondered to myself. 'Maybe it needs to rest a while.' I turned off the fan and lights and went back to finish my dinner. For the rest of the evening the fan and lights would turn on and off at random intervals and random levels, so I gave up on the idea that it would self-correct. With a heavy heart I went over to the breaker panel and flipped the hood fan breaker to and fro. The hood fan was once again well-behaved.

What's the embedded world coming to? Will programmers and companies everywhere realize the cost of their mistakes and clean up their act? Or will the entire world become accustomed to occasionally rebooting everything they own? Will expensive embedded devices come with a 'reset' button advertised as a feature? Will President Bush threaten war on programmers, unless U.N. code inspections are permitted? Or worst of all, will programmer jokes become as common and

ruthless as lawyer jokes? I wish I knew the answer. I can only hope for the best, but I fear the worst.

At last year's Embedded Systems Conference, a participant at a seminar I led admitted that his consumer products company could care less about the correctness of firmware. Reboot-who cares? Customers are used to this, trained by decades of desktop computer disappointments. Hit the reset switch, cycle power, remove the batteries for 15 minutes; even preteens know the tricks of coping with legions of embedded devices.

Crummy firmware is the norm, but it's totally unacceptable. Shipping a defective product is begging for torts. So far, the embedded world has been mostly immune from predatory lawyers, but that Brigadoon-like isolation is unlikely to continue. More importantly, it's simply unethical to produce junk.

But it's hard, even impossible, to produce perfect firmware. We must strive to make the code correct, but also design our systems to cleanly handle failures. In other words, a healthy dose of paranoia leads to better systems.

I was astonished to find that in almost 13 years of writing this column I've somehow managed to neglect watchdog timers. So here's my take, which is quite different from that espoused by most pundits.

Well designed watchdog timers fire off every day, quietly saving systems and lives without the esteem offered to human heroes.

Poorly designed WDTs also fire off a lot, sometimes saving things, sometimes making them worse. A simple-minded watchdog implemented in a non-safety critical system won't threaten health or lives, but can result in systems that hang and do strange things that annoy customers. No business can tolerate unhappy customers, so unless your code is perfect (whose is?), it's best in all but the most cost-sensitive apps to build a really great WDT.

An effective WDT is far more than a timer that drives reset. Such simplicity might have saved Clementine, but would it fire when the code tumbles into a really weird mode like the one experienced by Ed's hood fan?

What follows is a look at the current state of the art. Next month, I'll boil all of this down to concrete design recommendations.

Internal watchdogs

Virtually all highly integrated embedded processors include a built-in watchdog. Unfortunately, most of these are braindead.

Toshiba's TMP96141AF is part of their TLCS-900 family of microprocessors, which offers a wide range of extremely versatile on-chip peripherals. All have pretty much the same watchdog circuit. As the data sheet says, "The TMP96141AF is containing watchdog timer of Runaway detecting."

Ahem. And I thought the days of Jinglish were over. Anyway, the part generates a non-maskable interrupt (NMI) when the watchdog times out, which, as we'll see next month, is either a very, very bad idea or a wonderfully clever one. It's clever only if the system produces an NMI, waits a while, and only then asserts reset, which the Toshiba part unhappily cannot do. Reset and NMI are synchronous.

Motorola's widely-used 68332 variant of their CPU32 family (like most of these 68k embedded parts) also includes a watchdog. It's a simple-minded thing meant for low-reliability applications only.

Unlike a lot of WDTs, user code must write two different values (0x55 and 0xAA) to the WDT control register to ensure the device does not time out. This is a good thing; it limits the chances of rogue software accidentally issuing the command needed to appease the watchdog. I'm not thrilled that any amount of time may elapse between the two writes (up to the timeout period). Requiring two back-to-back writes would further reduce the chances of random watchdog tickles, though one would then have to ensure no interrupt could preempt the paired writes.

Another problem is that 0x55 and 0xAA patterns are often used in RAM tests; since the 68k I/O registers are memory mapped, a runaway RAM test could keep the device from resetting.

The 68332's WDT drives reset, not some exception-handling interrupt or NMI. This makes a lot of sense, since any software failure that causes the stack pointer to go odd will crash the code, and a further exception-handling interrupt of any sort would drive the part into a "double bus fault." The hardware is such that it takes a reset to exit this condition.

Motorola's popular Coldfire parts have a similar WDT. The MCF5204, for instance, will let the code write to the WDT control registers only once. Cool! Crashing code, which might do all sorts of silly things, cannot reprogram the protective mechanism. However, it's possible to change the reset interrupt vector at any time, pretty much invalidating the clever write-once design.

As with the CPU32 parts, an 0x55/0xAA sequence keeps the WDT from timing out, and back-to-back writes aren't required. The Coldfire datasheet touts this as an advantage since it can handle interrupts between the two tickle instructions, but I'd prefer a smaller window. The Coldfire has a fault-on-fault condition much like the CPU32's double-bus fault, so reset is the only option when the WDT fires-which is a good thing.

Unfortunately, there's no external indication that the WDT timed out, perhaps to save pins. That means your hardware/software must be designed so that at a warm boot, the code can issue a from-the-ground-up reset to every peripheral.

Philip's XA processors require two sequential writes of 0xA5 and 0x5A to the WDT. But, like the Coldfire, there's no external indication of a timeout, and it appears the watchdog reset isn't even a complete CPU restart; the docs suggest it's just a reload of the program counter. Yikes-what if the processor's internal states were in disarray from code running amok or a hardware glitch?

Dallas Semiconductor's DS80C320, an 8051 variant, has a powerful WDT circuit that generates a special watchdog interrupt 128 cycles before automatically, and irrevocably, performing a hardware reset. This gives your code a chance to safe the system and leave debugging breadcrumbs behind before a complete system restart begins. Pretty cool.

External watchdogs

Many of the supervisory chips we buy to manage a processor's reset line include built-in WDTs.

Texas Instrument's UCC3946 is one of many power supervisor parts that do an excellent job of driving reset only when Vcc is legal. In a small 8-pin surface mount package, it eats practically no real estate on a printed circuit board. It's not connected to the CPU's clock, so the WDT will output a reset to the hardware fail-safe mechanisms even if there's a crystal failure. But it's too darn simple: to avoid a timeout, just wiggle the input bit once in a while. Crashed code might do this in any number of ways.

TI isn't the only purveyor of simplistic WDTs. Maxim's MAX823 and many other versions are similarly flawed. The catalogs of a dozen other vendors list equally dull and ineffective watchdogs.

But both TI and Maxim offer more sophisticated devices. Consider TI's TPS3813 and Maxim's MAX6323. Both are "Window Watchdogs." Unlike the internal versions described above, which avoid timeouts using two different data writes (like a 0x55 and then 0xAA), these require tickling within certain time bands. Toggle the WDT input too slowly, too fast, or not at all, and a timeout will occur. That greatly reduces the chances that a program run amok will create the precise timing needed to satisfy the watchdog. Since a crashed program will likely speed up or bog down if it does anything at all, errant strobing of the tickle bit will almost certainly be outside the time band required.

The subject of watchdog timers is far too big to cover in a single column, so I'll carry on next month. But for now, remember that since software crashes are common, we're foolish not to equip our devices with some sort of recovery mechanism. And it's unwise to rely on a recovery strategy that's easily defeated by runaway code that could be doing virtually anything. When systems fail, they often do it in the worst possible way.

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.